Refactoring and Redesigning a Multidimensional, Open-Source Web Application Development System Using a Relational Database to Enhance Adaptability

Author: Jacques Levin, presenting the paper at the conference
Address: Nova Southeastern University, Office of Information Technologies and Media Services, Innovation Zone, 3301 College Avenue, Fort Lauderdale Florida 33314, email: jclevin@nsu.nova.edu, tel: 954-262-4883, fax: 954-262-2288

Author: John Scigliano, Professor
Address: Nova Southeastern University, Graduate School of Computer and Information Sciences 3301 College Avenue, Fort Lauderdale Florida 33314, email: scigl@nsu.nova.edu, tel: 954-262-2081, fax: 954-262-3915

Keywords: refactoring, multi-dimensional arrays, web server, relational database, RAD, Perl

Abstract:

The purpose of the authors in this paper is to address a problem and answer an important question. The problem was the large amount of code that existed in a legacy Web Application. The code count of the program made the system difficult to maintain and modify. In addition, navigating among the various screens in the Web Application was laborious and time consuming. The question was: Is it feasible, when refactoring and redesigning an application program based on given specifications, to keep the development specifications in a database, so that the program code would not have to have knowledge of the specifications (except for the basic architecture of Web applications)? If it is possible, then the program code - reduced to a Web Application Design Server [WADS] could act as a service for the design and development specifications. Our goal in the project was to refactor and redesign a Web Application Server. One main benefit of the WADS project was to reduce the effort involved in developing a Web application program by storing its specifications in a database format. All applications require modifications in time (some minor, some major), and that is a very time consuming process. The other main benefit of the WADS project was to provide an easy way to modify the application once it was developed, without having to modify the code of the WADS. In this article, the authors report on the design of this project and how it led to the development of the refactored version called RAD4Web.

1. Introduction:

The initial WADS project started after spending five years developing a fairly complex distance learning application called “ClassLeader” [1], consisting of some 60 screens organized hierarchically. In order to improve the system to make it apply to a wider range of application areas, we ran into problems. While making changes in some of the screens we realized that the size of the system code made it difficult to modify. Because of this we decided to study the source code more closely. Our reasoning was as follows: if we could refactor the 60 subroutines generating the 60 forms into one subroutine that could read data from a data file representing each
screen, modifying the system in the future would become much easier. William Grosso said that “in general, refactoring is modifying the code base to ensure that each thing is done ‘once and only once’. If something is done multiple times, each of those times should be replaced by a call to the one routine that does the thing well and doesn't do anything else. If closely related things are done at places scattered throughout the code base, they should be pulled together into a single routine” [2]. In our first attempt with the ClassLeader system, we had been able to produce a highly adaptable program, as long as the application fell within the realm of distance education. Encouraged by the results, we then decided to take this process a step further. We developed a general purpose web development server that was able to read the design specifications from data stored in a relational database. This system was then able to work with any Web application development project, not only for distance education. We called the final system RAD4Web [3].

2. Problem Statement:

The problem addressed in this project was the large amount of code that existed in the legacy WADS. The code count of the program made the system difficult to maintain and modify. In addition, navigating among the various screens in the earlier WADS was laborious and time consuming. This was fixed to increase the adaptability of the system and keep it competitive.

3. Goal:

Our goal was to create a second generation Web Application Development Server to reduce the effort involved in developing Web applications, navigating among screens, and improving the adaptability of the system.

4. Refactoring Process:

During the refactoring process, we focused on reducing the numerous subroutines into one unique subroutine. For that unique subroutine, we developed a simple algorithm to read the data for the form in four unique data lists or elements: field headers, field names, form elements, and values. These were preceded by four pointers to help separate the four lists. In this first development, we used two colons to separate each data element from the next. We then stored an entire form description on one line of the data file to be read by the unique subroutine. This process made any modifications that had to be made at a later time much easier, and it also reduced significantly the length of the program.

Since another one of our goals was to make changes in the process of navigating from screen to screen, we decided to continue the refactoring process to accommodate that. We realized at that time the navigation pattern was embedded in the source code in many statements involving the name, value pairs provided by the CGI protocol. We decided to proceed in two steps. In the first step, we created a unique subroutine controlling the entire navigation for the application. In the second step, we refactored the subroutine to read the navigation data stored in a data file, using one line for each navigation path. At the end of the process, we had reduced the size of the program in half, and were able to modify the application by simply modifying data files. Furthermore, there was no need anymore to modify the source code when making changes in an application.

5. Three-tier Architecture:

We developed a three-tier implementation (Figure 1) that included the following:
a) on the client side, a hierarchy of screens is displayed by sending back the output of a Perl program that was running on an Apache Web server

b) on the server side, a Perl program received as input the data submitted by a form, reading and writing data from the server and interfacing via DBI to a database

c) a relational database management system (MySQL) to store specifications for the Web application program

The Apache HTTP server is a popular Web server with more than 65% of the Internet Web sites using it [4]. Perl was developed over a decade ago, and has evolved into a powerful but simple to use, object-oriented language that works with databases using a DBI module (Database Interface Module) and CGI (Common Gateway Interface) [5]. MySQL is an open-source database that was developed in Sweden [6]. It has become one of the most widely used databases for the Web.

Figure 1. A three-tier architecture

6. Redesign -- Software Organization:

Starting from the general organization of a web application, in which users are prompted to enter information on a set of interconnected screens, we created an architecture comprised of six steps.

The six steps (Figure 2) included a hierarchy of Web pages that were viewed as a Web application. Each one of the Web pages was a collection of forms. Each form was designed so that if could be inserted on the graphic interface at a precise location. Any form could then be mapped into a relational table. Finally, the pages were linked to enable navigation.

Step 1. A Web application as a hierarchy of Web pages
In this system, a Web application is a hierarchy of Web pages that can be accessed in sequence as selections are made. Each selection takes the developer to the next level, each level consists of 1,2,3,...n pages.

Step 2. Each Web page as a collection of forms:
A web page may include any or all of the following: a standard menu, a feedback form, pull down menus, checkboxes or more generally entire feedback forms.

Step 3. Each form is made of form elements:
Step 4. Each form on a graphical interface:
To locate the forms (as a whole) or the form elements (field by field), we used an XML-like syntax, with the tag: <@ followed by a single character to identify the form or the field to insert at each location on the graphic interface (Figure 3).

```
| SCREEN 1 | <html>
| ...hypertext...
| title -------------- <@T-->
| image -------------- <@N-->
| menu -------------- <@M-->
| banner------------- <@B-->
| selectform-------- <@I-->
| submit menu------ <@H-->
| table display------ <@D-->
| form ------------- <@F-->
| radio display----- <@R-->
| footer----------- <@P-->
| image ----------- <@Q-->
|</html>
```

Figure 3: Graphical Interface
A collection of tags is used to mark the location of a variety of forms, including text near the top of the page (<@T), image near the top of the page (<@N), a menu (<@M), a banner, i.e. an horizontal menu (<@B), a select form, i.e. a form used for selection (<@I), a submit menu, i.e. a menu accompanied with data submission (<@H), a table display, i.e. a table of records displayed on the screen (<@D), a form (<@F), a radio display, i.e. a table of records preceded by radio buttons (<@R), a text near the bottom (<@P), and an image near the bottom (<@Q).

Step 5. Each form mapped into a relational table, to save the design in small tables (Figure 4).

<table>
<thead>
<tr>
<th>dir</th>
<th>to identify the application</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>the table name</td>
</tr>
<tr>
<td>field</td>
<td>the field name</td>
</tr>
<tr>
<td>formtag</td>
<td>the form element tag</td>
</tr>
<tr>
<td>length</td>
<td>the length of the field</td>
</tr>
<tr>
<td>fieldtype</td>
<td>the type of the field</td>
</tr>
<tr>
<td>select</td>
<td>to retrieve data from a foreign table</td>
</tr>
<tr>
<td>update</td>
<td>to update a foreign table</td>
</tr>
<tr>
<td>constraint</td>
<td>to specify a constraint relation</td>
</tr>
<tr>
<td>v01</td>
<td>to insert a first value</td>
</tr>
<tr>
<td>...</td>
<td>etc..</td>
</tr>
</tbody>
</table>

Figure 4: table representing one of the forms

Each form, and its equivalent table in the database, is parameterized to provide, for each field, its functionality, i.e: the name of the application directory (dir), table name (table), field name (field), the HTML tag (formtag) representing the form element (textfield, textarea, options, radio element, checkbox,...), field length (length), field type (fieldtype) (a primary key, a required field,...), alias (to associate a foreign key), redirect (to correlate 2 fields in the same table), select (to retrieve data from a foreign table, and display it for example in a pull down menu), update (to update a foreign table), constraint (to specify a constraint relation), target (to specify a target query), default (to specify a default value), date (to format the current date), subpanel (to specify a subpanel), followed by a maximum of 80 records to be loaded at initialization time.

For example,
i) A menu can be viewed as a table with one field whose values are the menu entries.
ii) A feedback form can be viewed as a table with as many fields as form elements, using the name of the form element as the field name, and the value either entered or selected on the form as the value of the corresponding field.
iii) A report can be viewed as a table with as many fields as there are columns and the values displayed as the field values.

Furthermore, many functionalities characterizing the interactive web application can be captured as table properties or attributes. For example:
i) The data displayed in a pull down menu, radio buttons or checkboxes may be automatically retrieved via a foreign key from a foreign table containing the data to be displayed.
It is also easy to add as a field property:
ii) Data characteristics, expressing the fact that a data item is a key element, or a required element, or cannot be changed
iii) Data restrictions, expressing the fact that a user may only view or select his/her own records
iv) Data constraints, expressing the fact that the data must be restricted to a given date, or a given category.

Step 6. Pages linked via a navigation mechanism:
The purpose of the navigation table was to provide, for each user selection, a mechanism that determines uniquely the next screen to be displayed (Figure 5).

When a user clicks on a button or selects a menu item on a page, the CGI protocol transmits that information from the client to the server using a set (that we call the path to the next screen) of name-value pairs, where name is the name of the selected HTML element, and value is the selected or entered value. The navigation table therefore includes, as primary key, the path to the next screen, and as other fields, a description of the content of the selected screen in terms of forms to be displayed.

We use a simple mechanism to separate the fields (a separator: =>), and to separate several forms to be displayed on the screen (a separator: ##). Each form name is preceded by a form type represented by a single character (M for menu, F for form, etc.). The RAD4Web uses this character to locate the form on the graphic interface. Each record is preceded by a navigation level. For example, the first row of the navigation table: 0: => Mstartlist means that, when starting the application, the system will display a menu named startlist on the graphic interface at the location marked by the tag <@M). This menu will be further described by a table called startlist represented by the table structure shown in Figure 4.

```
0: => Mstartlist
  1: charter => Mcharterlist
    2: schedule => Hschedulelist ## Fschedule
    3: chartercust => Fchartercust
  2: edit => Islectchart
    3: * => Rschedule
  4: * => Hschedulelist ## Fschedule
    5: chartercust => Fchartercust
    2: manifest => Iselectchart
```

Figure 5: Navigation table

7. Redesign -- Rapid Application Development for the Web (RAD4Web):

The design in this project involved two versions of RAD4Web. The difference between these RAD4Web versions consisted of a major improvement in Version 2 by using Perl multi-dimensional arrays. The arrays provided a data structure equivalent to relational tables in a database. That meant that the development specifications stored in the database could be read directly into the Perl multi-dimensional arrays. The change was sufficiently powerful to reduce significantly the size of the code between the RAD4Web Version 1 and Version 2.
7.1 RAD4Web Version 1. Several applications were developed in Version 1, and these were presented in a previous paper [7]. In our first version of RAD4Web, we used the standard Perl language, the CGI protocol and the DBI interface working with a MySQL database. These components were used to generate nine distinctly different web applications. However, these applications did not use multidimensional arrays.

7.2 RAD4Web Version 2: Encouraged by these results, we then developed a second version, using Perl modules and multidimensional arrays (Arrays of Arrays, Hashes of Arrays, Arrays of Hashes and Hashes of Hashes) [8]. In this version, we created Perl relations equivalent to the relational tables stored in the database. This led to a further reduction in the size of the program (from 50,000 lines in the original ClassLeader program to 5,000 lines in RAD4Web Version 2). Meanwhile, we were able to maintain the readability of the code by a parameterization of the design.

8. Conclusions:

Two years ago, we started a Rapid Application Development project for a Web server using open source software, mainly an Apache server, the Perl language and the MySQL database. We then extended the project to other Web servers (Microsoft IIS, ) and other databases (Oracle, Microsoft SQL server, …). In this paper, we have presented a process for refactoring and redesigning the Web Application Development Server, and this led to a new system called RAD4Web. The Open Source software enabled us to design a system that could work in several different application development areas. Through refactoring the system was made more adaptable, took significantly less time to maintain, and could be easily modified for future applications.

References:

http://www.thejournal.com/magazine/vault/A2881.cfm, Retrieved February 3, 2004

Retrieved February 3, 2004


Retrieved February 3, 2004

http://www.perldoc.com/perl5.6/pod/perldsc.html Retrieved February 3, 2004